PIA Programmation Impérative Avancée (Advanced Imperative Programming)

Stéphane Genaud — Jens Gustedt

March 7, 2016

Outline

Table of Contents

Organization and Schedule

- UE PIAL (Programmation Impérative Avancée et Logique)

Module	Evaluation	date	Module weight	UE weight
PIA	TP noté	march 16	1/3	
	Written Exam	march 23	2/3	1/2
MLO	Written Exam	may 18	1	1/2

Table of Contents

Modularity

Necessary to split the code base in parts as it grows

- parts are often called modules

Useful to:

- logically structure the programs,
- reuse modules
- share the development process between programmers

How to structure?

- Separate "logical" entities in different modules.
- Example: an abstract data type (ADT) and its implementation(s)

Modules

A module includes

- Data Structures
- Functions to manipulate these data structures

- How to "link" the module to the rest of the code : interface
  - specify the functions exposed = *contract *

Outside the module, only elements declared in the interface might be used (black box).

Interface

Interface

An Interface is a contract for using what the module provides.

An interface contains:

- type definitions
- functions prototypes

Interface in C

In C, no enforcement to use interfaces, but conventions:

- interfaces are in *headers files*, e.g `stdio.h`
- interfaces contain function prototypes and type definitions

```
typedef struct __sFILE FILE;
int fseek(FILE *, long, int);
```

Example: display a number in base 2

- The algorithm we want to implement proceeds by successive divisions of N to obtain the digits d_i
  1. divide N by 2, N mod 2 is the last digit
  2. N $\longleftarrow \lfloor N/2 \rfloor$
  3. loop to step 1 to find the previous digit, until first digit

Example: display a number in base 2

- We want to display the successive division remainders in reverse order

Example: display a number in base 2

- We want to display the successive division remainders in reverse order
- A convenient data structure is the stack (LIFO)

Example: display a number in base 2

- We want to display the successive division remainders in reverse order
- A convenient data structure is the stack (LIFO)
- Let assume the following interface

```
typedef struct stack_base* stack;
stack new();
bool is_empty(stack);
void push(stack*, int);
int pop(stack*);
```

Example: display a number in base 2

- Without knowing the implementation, we can write our algorithm:

```
void print_base2(int n) {
   int i;
   stack p;
   p = new();
   while (n > 0) {
       push(&p, n%2);
       n=n/2;
   }
   while (!is_empty(p)) {
       i = pop(&p);
       printf("%i",i);
   }
}
```

Abstract Data Type

Does a user of the Stack library need to know how it is implemented to use it?

- Answer: No. He/she just need to know the primitives available to handle such stack "objects".

Definition (Abstract Data Type – ADT)

A set of data values and associated operations that are precisely specified independent of any particular implementation

Abstract Data Type Example : Queue .

Values

- Queue, Element, boolean

Operations

create	:		$\rightarrow$	Queue
enqueue	:	Queue x Element	$\rightarrow$	Queue
dequeue	:	Queue	$\rightarrow$	Element
isempty	:	Queue	$\rightarrow$	boolean

Axomiatic (specification)

- isempty(create()) = true
- isempty(enqueue(Q,e)) = false
- dequeue(create()) = ϵ
- dequeue(enqueue(Q,e)) = e

Choice 1 : Expose structs

foo.h

```
typedef struct concrete_type {
     int x;
     double y;
} concrete_type;

void bar(concrete_type *,int);
```

foo.c

```
#include "foo.h"

void bar(concrete_type *t, int y) {
    .....
    t->x = y;
    .....
}
```

main.c

```
#include "foo.h"

conrete_type *t = malloc(sizeof(conrete_type));
t->x = 1; /* ok */
```

Choice 2 : Make struct OPAQUE

foo.h

```
typedef struct opak abstract_type;

void bar(abstract_type *,int);
```

foo.c

```
#include "foo.h"
struct opak {
     int x;
     double y;
};

void bar(abstract_type *t, int y) {
    .....
}
```

main.c

```
#include "foo.h"

abstract_type *t = malloc(sizeof(abstract_type)); /* ! incomplete type */

t->x = 1; /* ! Will not compile ! */
bar(t,1); /* OK, manipulate the DS through the interface */
```

Dependencies

- A module depends upon
  - its implementation,
  - the modules' interfaces it uses
- If these change, the module must be recompiled.
- A module does not depend upon the modules' implementations

do not need to recompile the module if they change.
-It becomes quickly difficult to remember what must be recompiled:
a Makefile enables to track the dependencies.

Separate Compilation

Compilation produces an object file

```
gcc -c tree.c #--> produces tree.o
gcc -c tree-main.c #--> produces tree-main.o
```

External references (e.g external functions) are not resolved

Linking of object files produces an executable

```
gcc tree.o tree-main.o -o tree-main
```

It is checked that external references can be found among object files

Tools to handle separate compilation

- make: the most commonly used tool in the last decades.
- ant: mainly used in Java Projects.
- Other tools have build upon Makefiles: GNU autotools, CMake.

Makefile

- a `Makefile` is processed by `make`
- A set of rules of the form

```
target: depends_1 depends_2 ... depends_n
        command
```

(A `tab` char is mandatory at the start of the 'command' line.)

1. `target`: the file to be created
2. `depends_i`: the files upon which `target` depends
3. `command`: often, the command used to produce `target`

```
tree-example: tree.o tree-example.o
        gcc tree.o tree-example.o -o tree-example
```

Makefile: rule trigger

```
target: depends_1 depends_2 ... depends_n
        command
```

`command` is invoked if

- some `depends_i` is more recent than `target`
- or if `target` does not exist

Makefile: rule trigger

```
main.o: main.c foo.h
	gcc -c main.c

foo.o: foo.c foo.h
	gcc -c foo.c

prog: main.o foo.o
	gcc main.o foo.o -o prog
```

Makefile: rule trigger

- Several rules: `make` applies the first rule found.
- The convention is to call it `all`.
- Triggers the rules following the dependency chain.

```
all: tree-example queue-example

tree-example: tree.o tree-example.o
	gcc tree.o tree-example.o -o tree-example

tree-example.o: tree-example.c
	gcc -c tree-example.c

queue-example: queue.c
	gcc queue.c -o queue
```

Makefile: variables

Variables (aka macros) can be defined:

- From the environment

```
export CC=gcc
make
```

- In the Makefile

```
CC=gcc
OBJS = main.o sub1.o sub2.o sub3.o
...
$(CC) $(OBJS) -o main
```

Makefile: variables

Using Variables

- Variable `A` is used with `$(A)`
- They can be assigned only once

Predefined Variables

`$@`	target name
`$?`	the dependent names younger than the target
`$^`	dependency name
`$<`	the dependency which triggered the action

Makefile: substitutions

Lists can be build from substitutions

```
SRC=tree.c queue.c tree-main.c
OBJ=${SRC:.c=.o}
```

will make `OBJ = tree.o queue.o tree-main.o`

Makefile Implicit Rules

Implicit rules

A rule of the form:

```
s1s2 :
	commands to get s2 from s1
```

Allowed suffixes are given by the `.SUFFIXES` fake target

```
.SUFFIXES: .o .c

# define a suffix rule for .c -> .o
.c.o :
	$(CC) $(CFLAGS) -c $<
```

See list of implicit rules from GNU

makedepend tool

- Automatically finds the dependencies for each object file
- Scan (recursively) the `#include` directives

Table of Contents

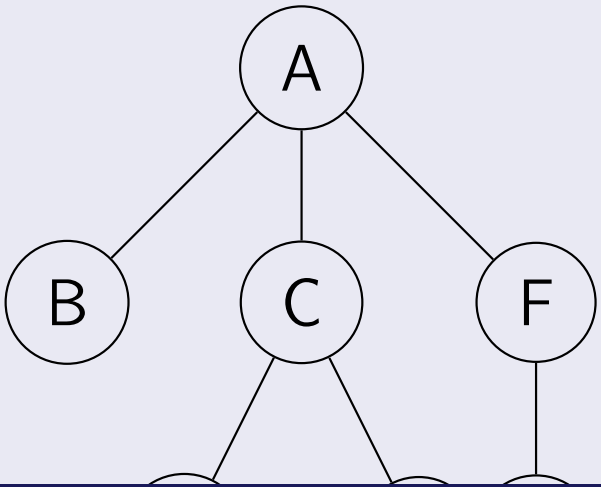
Definitions

A tree

- An acyclic and undirected graph with a single root.
- $\Rightarrow$ any pair of vertices is connected by exactly one path

Vertices, aka *Nodes*

- the root (la racine)
- the internal nodes (les noeuds internes)
- the leaves (les feuilles)

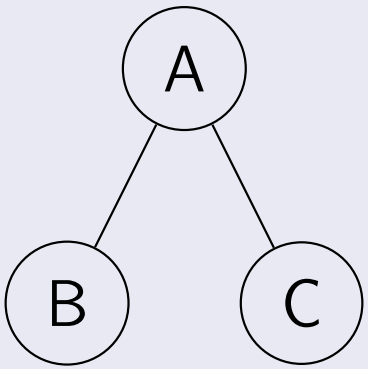


Binary Trees

Binary Tree

Each node has at most two children

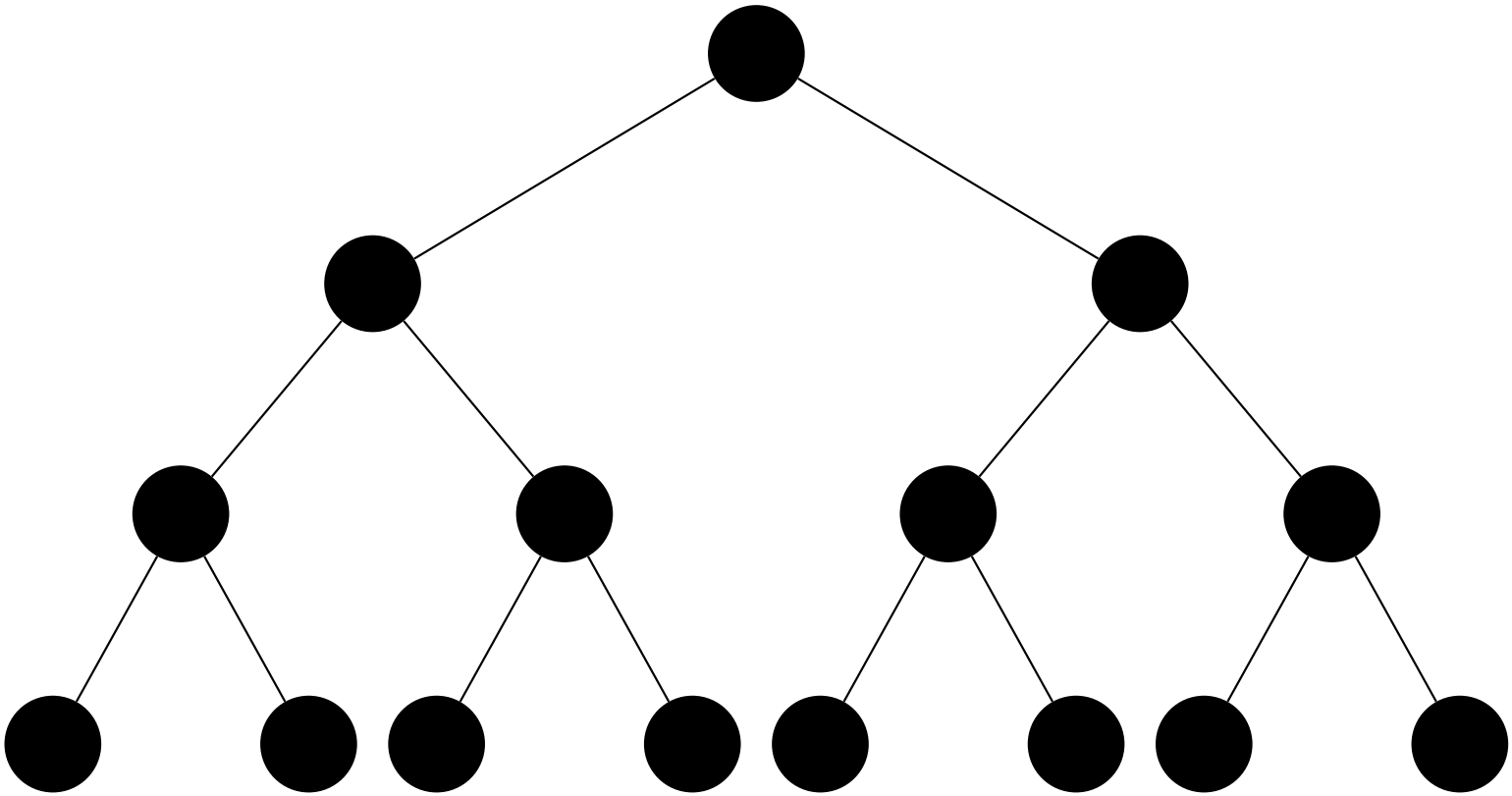
- useful for dichotomic purposes (e.g Search Tree)
- (not to confound with *B-trees* which stands for Balanced Trees)



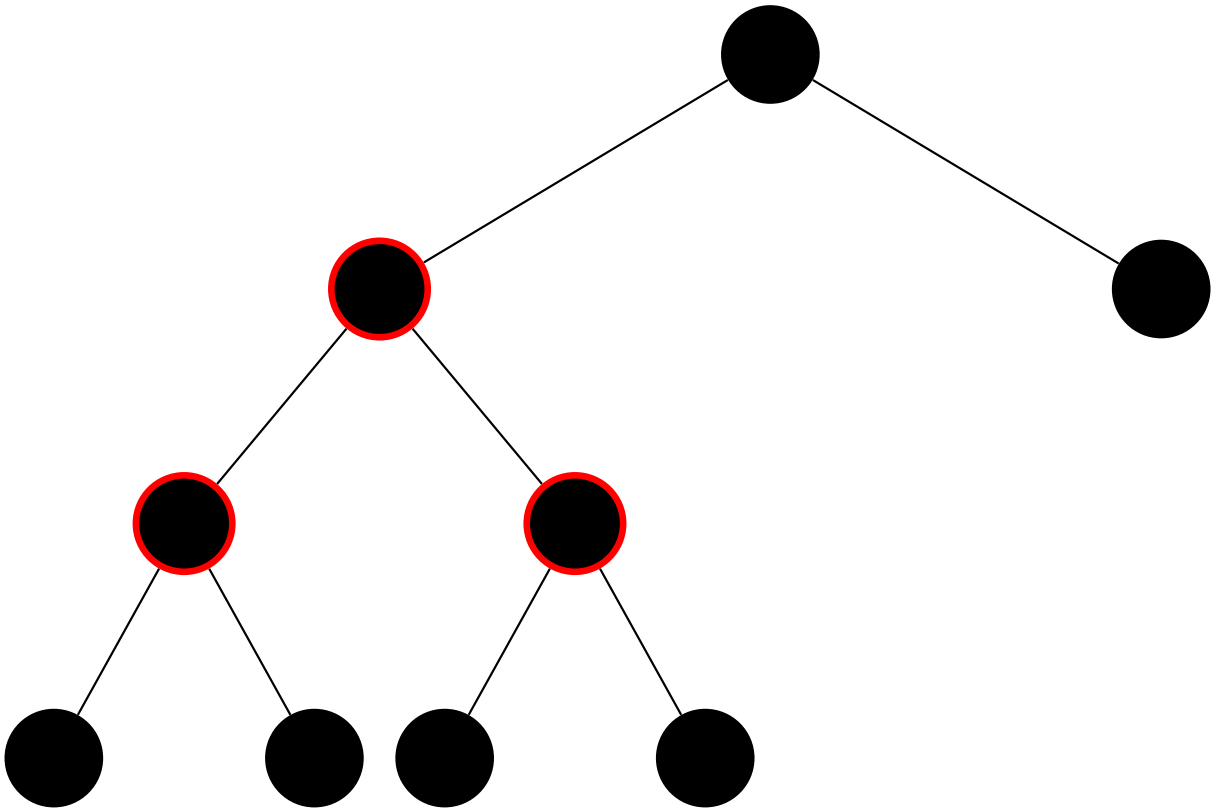
Definition

- size of a tree: the number of nodes.
- height of a tree: the number of nodes on the longest branch from the root to a leaf.
- a binary tree is complete if all its leafs are at the same distance from the root.
- a binary tree is locally complete if all its internal nodes have exactly 2 children.

Example complete tree

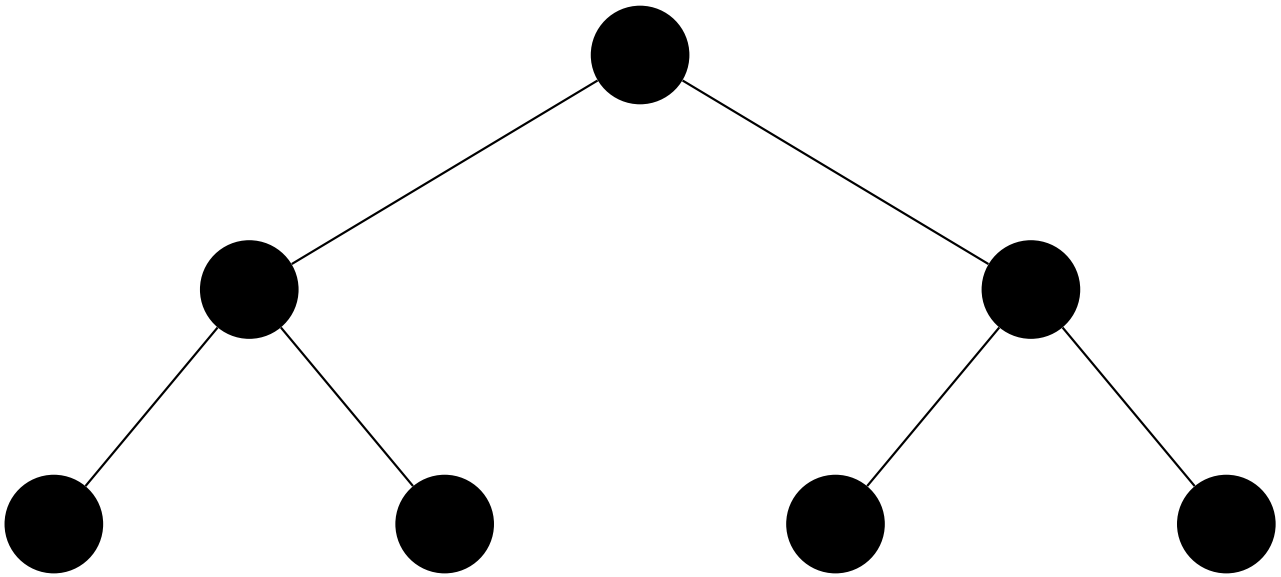


Example locally complete tree



Properties of a complete tree

- A level: the nodes at the same distance from the root.



- Number of nodes at a given level L ($L \in \{1, \ldots, n\}$)

$$N = 2^{L-1}$$

Properties of a complete tree

- Number of nodes of a complete tree (i.e its size) with height h is

$$S = \underbrace{1 + 2^1 + 2^2 + \cdots + 2^{h-1}}_{h \text{ terms}}$$

- Sum of a geometric progression with scale factor q, n terms, intial term u_0 is $S = u_0 \frac{1-q^n}{1-q}$.
- Hence, the size of a complete tree is

$$S = 1 \cdot \frac{1 - 2^h}{1 - 2^1} = 2^h - 1$$

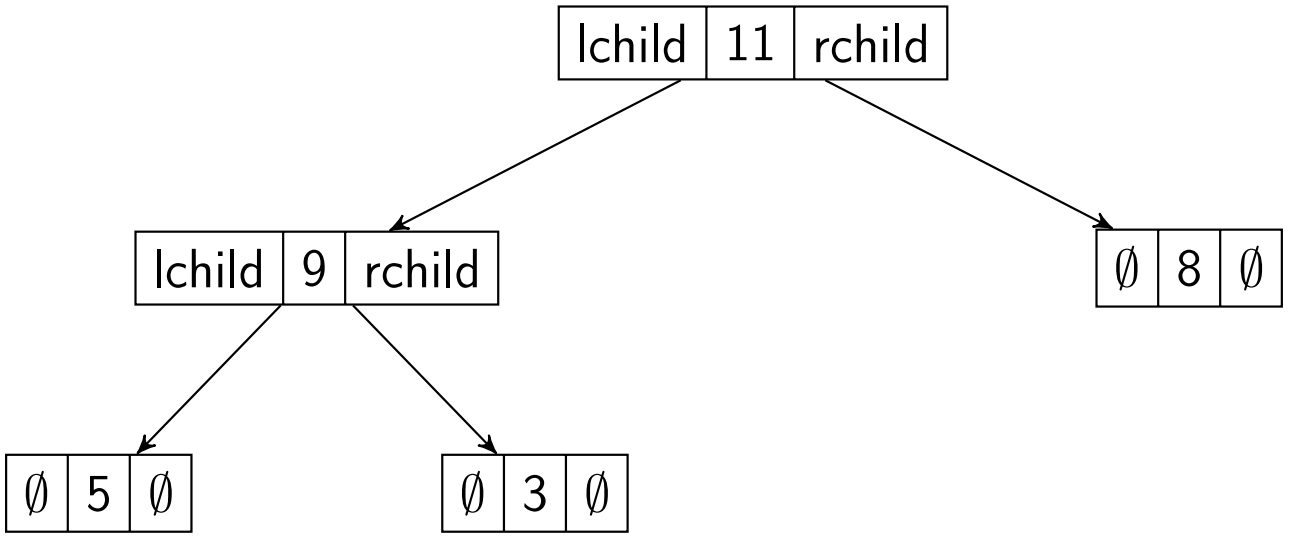
Table of Contents

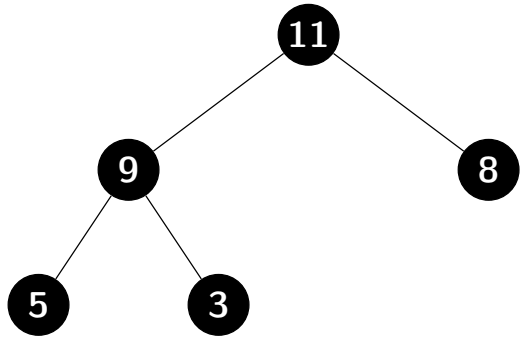
Categories

We need to:

- Create trees
- Change the tree structure
- Query the tree structure
- Navigate into trees

Design a Data Structure





C Implementation with pointers

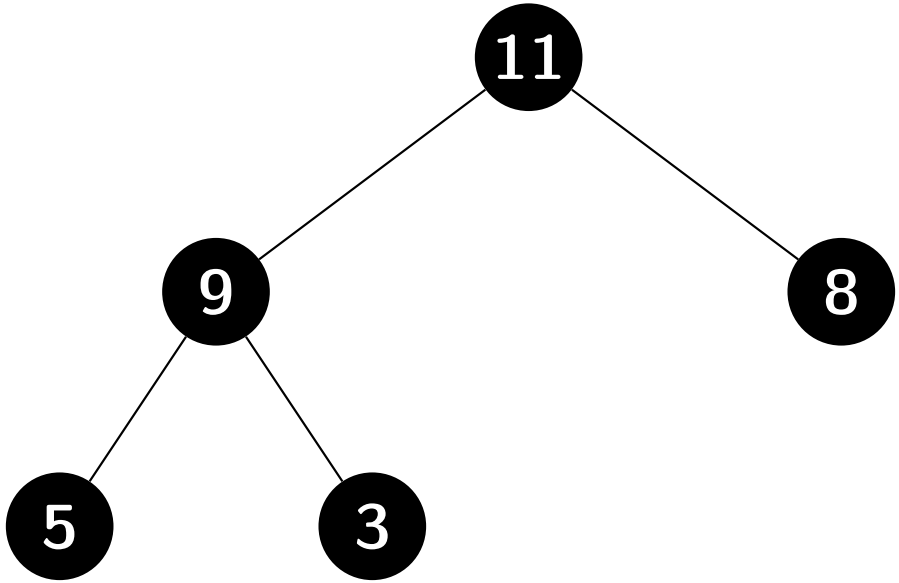
```
struct node {
        int value; /**< one value per node */
        node *lchild; /**< the left child */
        node *rchild; /**< the right child */
};
```

```
typedef struct node node;
```

C Implementation as an Array

index	0	1	2	3	4
value	11	9	8	5	3

$$left(i) = 2i + 1$$
$$right(i) = 2i + 2$$
$$parent(i) = \lfloor i/2 \rfloor$$



How to start to build the Tree library?

How would you design a `tree_create()` primitive?

- Maybe tell the size, or the height of the tree
- What does `tree_create()` return ?

Node

Base elements are nodes

- We need a `node_create()` primitive.
- User only cares about the value the node holds.

```
node* node_create(int val);
```

- Convenient to abstract the attachement of a node to another

```
node* node_insert(node *root, int val, direction dir);

enum direction { left, right };
typedef enum direction direction;
```

Tree Creation 1

A complete binary tree from its height

```
node *tree_create_from_height(size_t height);
```

- Algorithm `create_from_height`: Given a height h
  1. If $h = 0$ return an empty tree
  2. Otherwise, create a node N
  3. Invoke the creation of a tree of height $h-1$ rooted in N

Tree Creation 2

A binary tree from its size

```
node *tree_create_from_size(size_t size);
```

Note: *Size may not allow for complete tree*

- Algorithm `create_from_size`: Given a size s
  1. If $s = 0$ return an empty tree
  2. Otherwise, create a node N and update size: $s \leftarrow s - 1$
  3. split remaning nodes in two: $s_1 = \lfloor \frac{s}{2} \rfloor$ and $s_2 = s - s_1$
  4. Invoke the creation of a tree of size s_1 rooted as left child of N
  5. Invoke the creation of a tree of size s_2 rooted as right child of N

Tree Query: size

```
size_t tree_size(node *n);
```

- Algorithm `tree_size`: Given a root node *n*
  1. If *n* is empty return 0
  2. Otherwise, return 1 + `tree_size` of left child + `tree_size` of right child

Tree Query: height

```
size_t tree_height(node *root);
```

- Algorithm `tree_height`: Given a root node N
  1. If N is empty return 0
  2. Otherwise, return 1 + maximum between `tree_height` of left child and `tree_height` of right child

Node Query

Convenient functions

```
bool node_isinner(node *n);
bool node_isleaf(node *n);
```

Tree Traversal

Natural order is depth-first traversal

- may be in prefix, infix or postfix order.
- Naturally inductive: for any node, recursively traverse left child then right child.
- order refers to when is used the value (after, between, after the recursion). See below.

Often useful is breadth-first traversal

- Use a queue to enqueue nodes at a same level, and treat them in that order.

Depth-first Traversal (prefix order)

```
void tree_prefix_visit(node *n) {
    if (n) {
        /* do something with value */
        tree_prefix_visit(n->lchild);
        tree_prefix_visit(n->rchild);
    }
}
```

Depth-first Traversal (infix order)

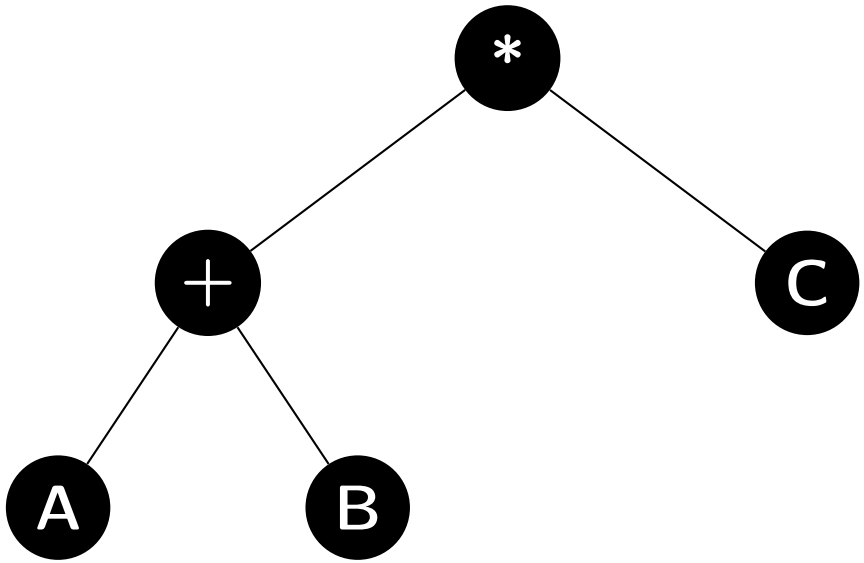
```
void tree_infix_visit(node *n) {
     if (n) {
          tree_infix_visit(n->lchild);
          /* do something with value */
          tree_infix_visit(n->rchild);
     }
}
```

Depth-first Traversal (postfix order)

```
void tree_postfix_visit(node *n) {
     if (n) {
          tree_depthfirst_visit(n->lchild);
          tree_depthfirst_visit(n->rchild);
          /* do something with value */
     }
}
```

Illustration of pre/post/infix orders

- prefix: * + A B C
- postfix: A B + C *
- infix: A + B * C



Breadth-first Traversal

```
void tree_breadthfirst_visit(node *n) {
    queue *q;

    while (n) {

       /* ... do something with n->value ... */

       if (n->lchild)
          queue_enqueue(q, n->lchild);
       if (n->rchild)
          queue_enqueue(q, n->rchild);

       n = queue_dequeue(q);
    }
}
```

Modularity

- We just created a *tree* library, functions and implementations
- We want to offer it as a module
- How should the interface be exposed?

C header files

tree.c

```
typedef struct node node;
struct node {
   int val;
   node *lchild;
   node *rchild;
 };

node* tree_create(size_t n) {
  if (!n) return 0;
  node* root=node_create(n);
   ...
}
node_create(node *n) {
....
}
```

What should i do to:

- make `tree_create()` public
- hide `node_create()`
- hide `node` struct

Choice 1 : Expose structs

tree.h

```
typedef struct node node;
struct node {
   int val;
   node *lchild;
   node *rchild;
 };
node* tree_create(size_t n);
```

tree.c

```
#include "tree.h"
node* tree_create(size_t n) {
  if (!n) return 0;
  node* root=node_create(n);
  ...
}
static node_create(node *n) {
....
}
```

All file that include `tree.h` can access the struc

Choice 2 : Make struct OPAQUE

tree.h

```
typedef struct node node;
node* tree_create(size_t n)
```

tree.c

```
#include "tree.h"
struct node {
   int val;
   node *lchild;
   node *rchild;
 };

node* tree_create(size_t n) {
  if (!n) return 0;
  node* root=node_create(n);
   ...
}
static node_create(node *n) {
....
}
```

Table of Contents

Program efficiency

- to assert the efficiency of programmes is difficult:
- What is a good algorithm?
- What is a good implementation?
- There is no overall answer to these questions:
  - depends on the use case
  - depends on the platform
  - depends on the investment costs

Example for both lectures: dictionaries

- abstract interface dictionary type:

```
dict* dict_create(void);
dict* dict_delete(dict* d);
size_t dict_size(dict const* d);
dict* dict_insert(dict* d, dict_key k, dict_value v);
bool dict_search(dict* d, dict_key k, dict_value* v);
dict* dict_remove(dict* d, dict_key k);
```

Algorithm complexity

- An *algorithm* is a finite sequence of elementary operations that constitutes a computation scheme or a problem resolution.
- The *time complexity* of an algorithm measures the number of elementary operations that are run on a given data set. It is expressed as a function T of the input size n
- The unit of measure is the "most significative" operation, *e.g*
  - addition/subtraction and/or
  - multiplications/divisions
  - memory access
  - data base queries
  - ...

Example

- complexity of bubble sort (advance the maximum element)
- elementary operation: comparison
- how many comparisons in total?

x_0	$\overset{?}{\longrightarrow}$	x_1	$\overset{?}{\longrightarrow}$	x_2	$\overset{?}{\longrightarrow}$	x_3	$\overset{?}{\longrightarrow}$	x_4	$\overset{?}{\longrightarrow}$	x_5
x_1	$\overset{?}{\longrightarrow}$	x_2	$\overset{?}{\longrightarrow}$	x_0	$\overset{?}{\longrightarrow}$	x_4	$\overset{?}{\longrightarrow}$	x_5		x_3
x_2	$\overset{?}{\longrightarrow}$	x_1	$\overset{?}{\longrightarrow}$	x_4	$\overset{?}{\longrightarrow}$	x_0		x_5		x_3

- First bubble up: $n-1$ comparisons
- Second bubble up: $n-2$ comparisons

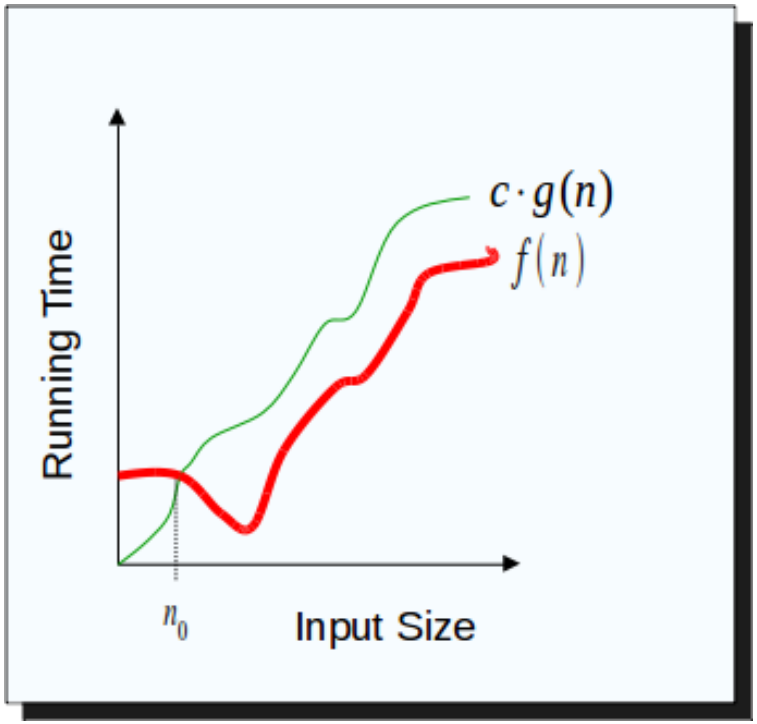
  ... n-1 times

  $\Longrightarrow \frac{n(n-1)}{2} = \frac{n^2-n}{2}$
- the complexity is quadratic

Asymptotic Behavior (Landau notation)

$f(n) = O(g(n))$ iff there exist $c > 0$ and $n_0 > 0$, such that for all $n \geq n0$

$$0 \leq f(n) \leq c \cdot g(n)$$



- Ex : the previous sort is in $O(n^2)$

Asymptotic Behavior (Landau notation)

- Simple Rule : we omit the terms of lower order and the constants
  - $50n \log n$ is in $O(n \log n)$
  - $7n - 3$ is in $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is in $O(n^2 \log n)$
- Remark: Altough $(50nlogn)$ is also in $O(n^2)$, or in $O(n100)$, we use the smallest possible order.

Several complexities for an algorithm

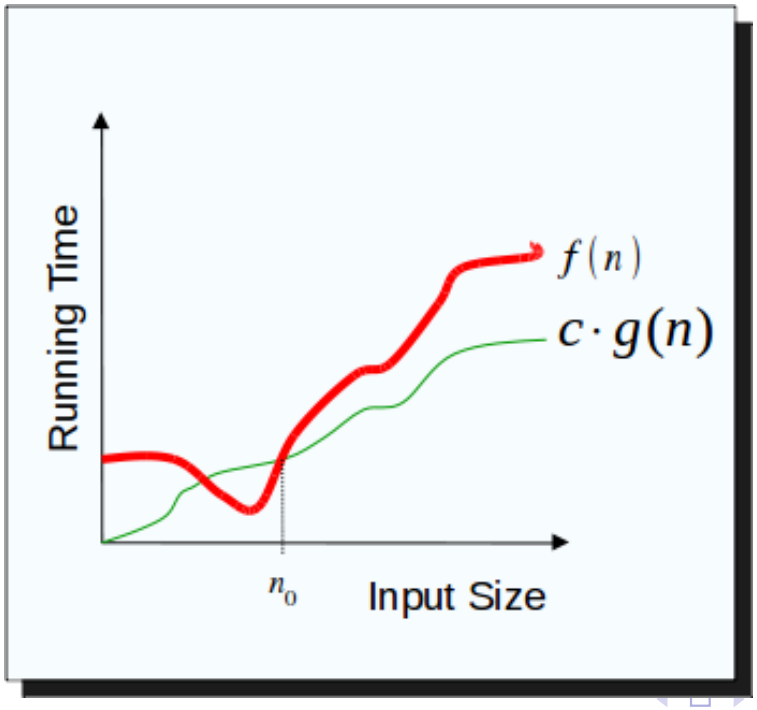
- best case complexity, worst case, average case
- e.g: search in a sorted array:
  - best case: $O(1)$
  - worst case: $O(n)$
  - on average: $O(n/2)$ that is $O(n)$

Other notations: Ω-Notation

Lower asymptotic bound
$f(n) = \Omega(g(n))$ iff there exists $c > 0$ and $n_0 > 0$, such that for $n \geq n_0$

$$0 \leq c \cdot g(n) \leq f(n)$$

Used to indicate a best-case complexity



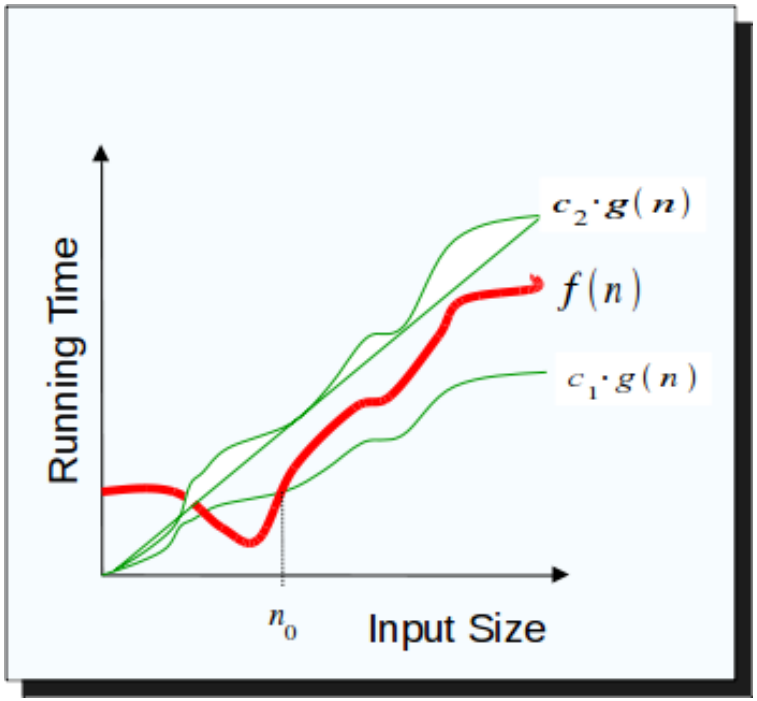
Other notations: Θ-Notation

- Lower and upper asymptotic bounds
- $f(n) = \Theta(g(n))$ iff
  - there exists constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$, such that for $n \geq n_0$

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Other notations: Θ-Notation

- $f(n) = \Theta(g(n))$ ssi
- $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$



- Remark : $O(f(n))$ is often used for $\Theta(f(n))$.

Some Figures

for usual orders of magnitude

$\backslash\ T(n)$ n	$\log_2 n$	$n \cdot \log_2 n$	n^2	n^3	2^n
10^1	3	$3.3\ 10^1$	10^2	10^3	10^3
10^2	7	$6.6\ 10^2$	10^4	10^6	$1.3\ 10^{30}$ (d)
10^3	10	10^4	10^6	10^9	
10^4	13	$1.3\ 10^5$	10^8	10^{12} (b)	
10^5	17	$1.7\ 10^6$	10^{10}	10^{15}	
10^6	20	$2.0\ 10^7$ (a)	10^{12} (b)	10^{18} (c)	

At 1 giga-flops (10^9 floating point operations/sec)

(a) 20 ms (b) 17 minutes
(c) 32 years (d) 40 billion years

Some examples

Insertion into a sorted list:	$\Theta(n)$
Insertion at the list head:	$O(1)$
Insertion at the end of the list (if no direct pointer):	$\Omega(n)$
Quick Sort:	$\Theta(n \log n)$
standard matrix multiplication:	$O(n^{3/2})$
multiplication of polynomials:	$\Theta(n^2)$
...	...
combinatorial problems:	$\Omega(2^{\alpha n})$

problem classes

- the size N of a problem P is the amount of information that is necessary to represent it in a formal language.
- Decidable and undecidable: we know (or not) that there is (or there is no) algorithm to solve P.
- Polynomial: there is an algorithm in time $O(N^k)$ to solve P
- NP: there is a polynomial algorithm to verify any solution to P
- NP Complet: there is an exponential algorithm (enumeration of all 2^N possibilités) to solve P but there is no polynomial algorithm.

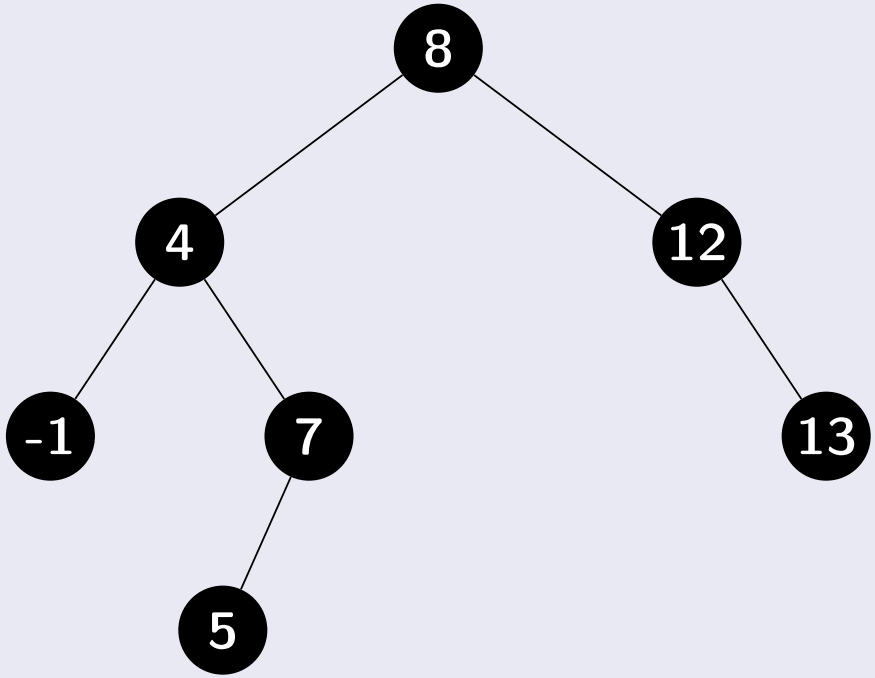
Table of Contents

Binary Search Trees

- Each tree node has stored a pair $(key, value)$.
- Search is done relative to *key* and retrieves a *value*.
- There is total order $(\leq)$ on the keys
- For each tree node n, all nodes within
  - the left subtree $x \in left(n)$ have $key(x) < key(n)$
  - the right subtree $x \in right(n)$ have $key(x) > key(n)$
- *exemple*: dictionary, keys are words, values are texts
- *goal*: speed up searching for keys
- *remark*: special care has to be taken if keys can be repeated

Binary Search Trees

Example: integer keys



Binary Search Trees

Insertion to a BST

The simplest way to implement *insert* a new pair (k, v) is to add a new node as a leaf:

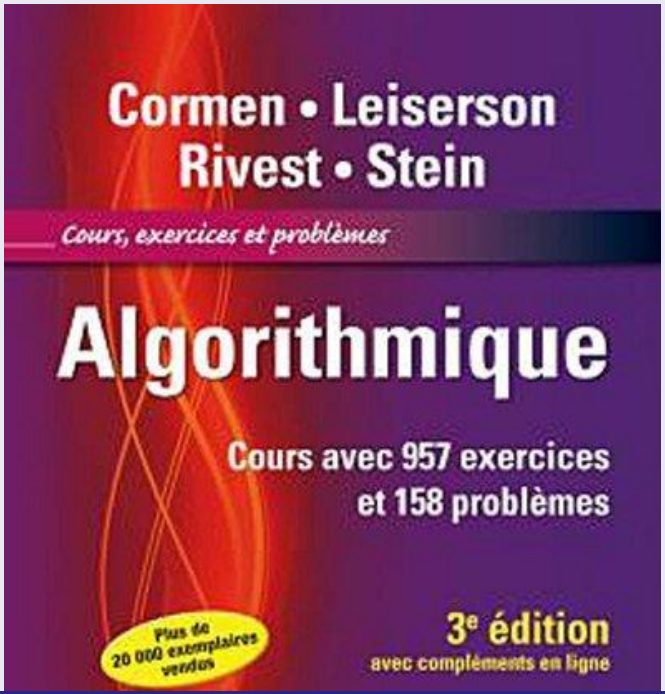
- Create a new leaf node n with
  - (k, v)
  - $left(n) = \emptyset$
  - $right(p) = \emptyset$.
- Search for a node p that has either
  - $k < key(p)$ and $left(p) = \emptyset$
  - $key(p) < k$ and $right(p) = \emptyset$
- Make n the appropriate child of p

Binary Search Trees

Complexity of the insertion:

best case:	$O(\log n)$	balanced tree
worst case:	$O(n)$	degenerated tree
average:	$O(\log n)$	see Cormen, non-trivial proof

Cormen, Leiserson, Rivest, Stein



AVL Trees

Goals

- Improve the worst case for insertion to $O(\log n)$.
- Achieve the same complexity also for deletion.

AVL Trees

Idea: balance a BST

G. Adelson-Velskii and E. M. Landis, An algorithm for the organization of information. Proceedings of the USSR Academy of Sciences 146: 263–266, 1962

- balance tree a each insertion/deletion, guarantee that

$$|h_{max} - h_{min}| = 1$$

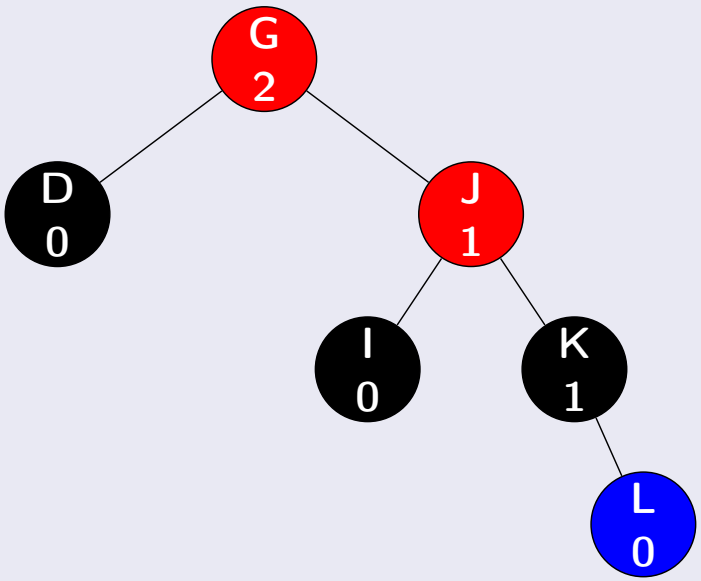
- in each node n compute imbalance

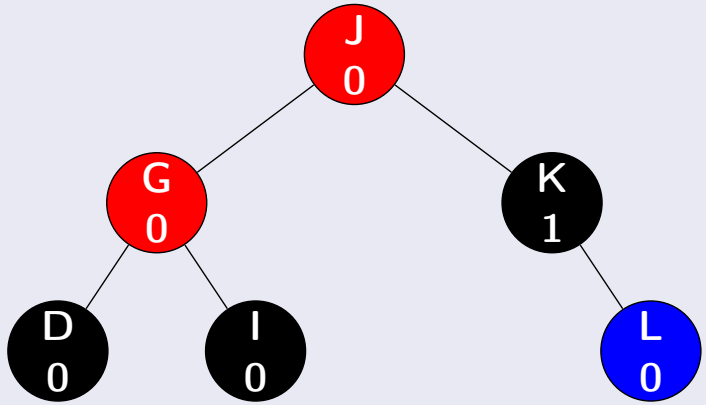
$$h_{right(n)} - h_{left(n)}$$

AVL Trees

rotation

- Insert node L
- Try to preserve balance

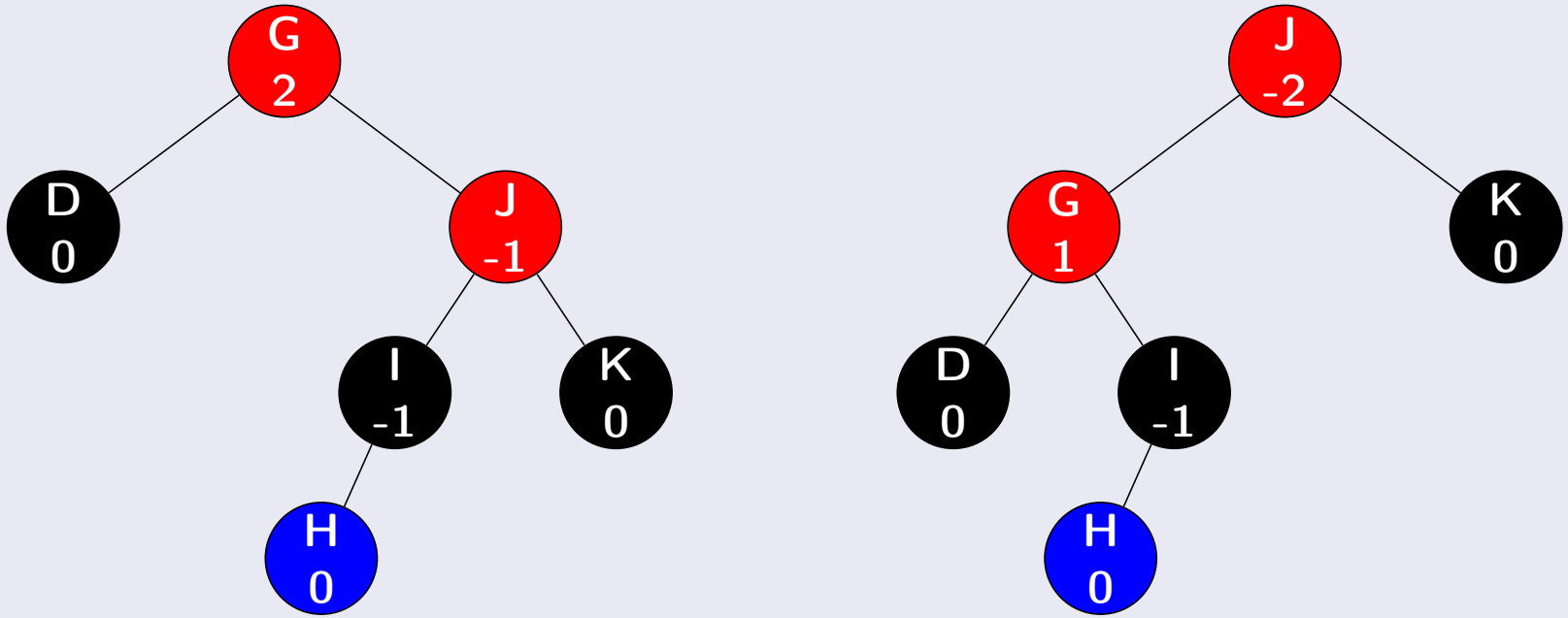




- To balance we move J upward: left rotation

AVL Trees

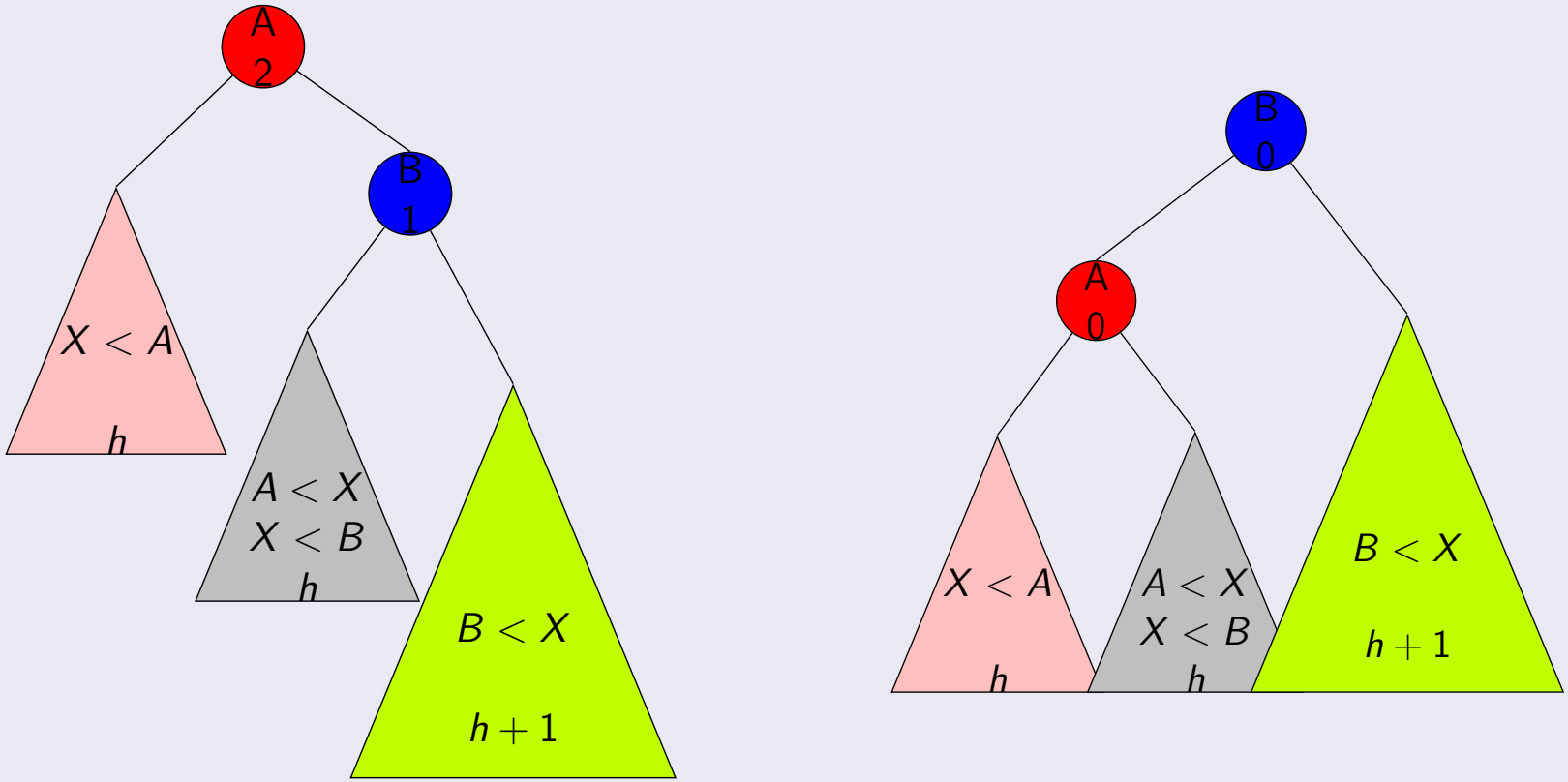
a simple rotation might not be sufficient



- Solution: double rotation, right-left

AVL Trees

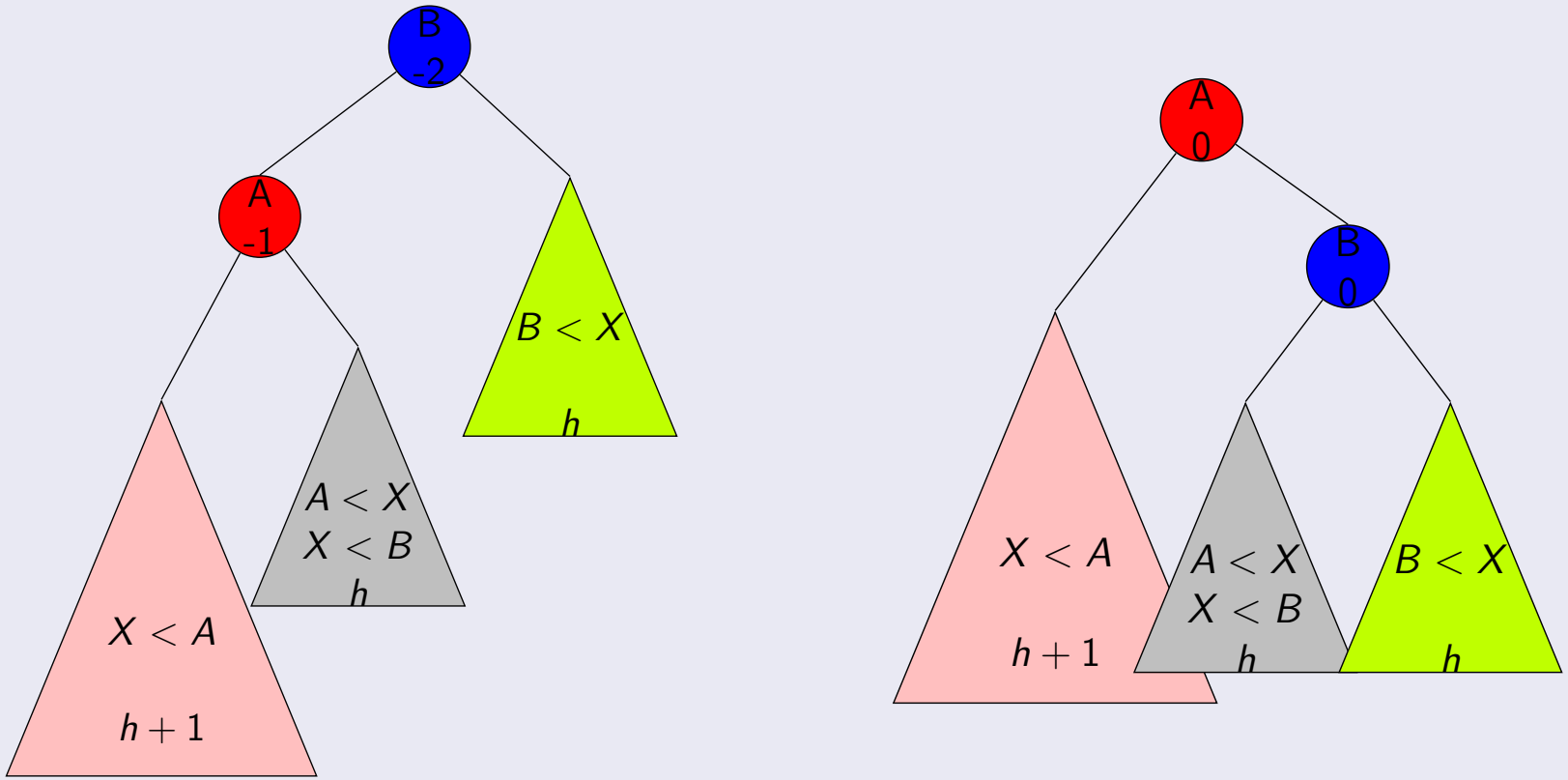
rotation 1/4: imbalance +2, imbalance right child +1



- simple left rotation

AVL Trees

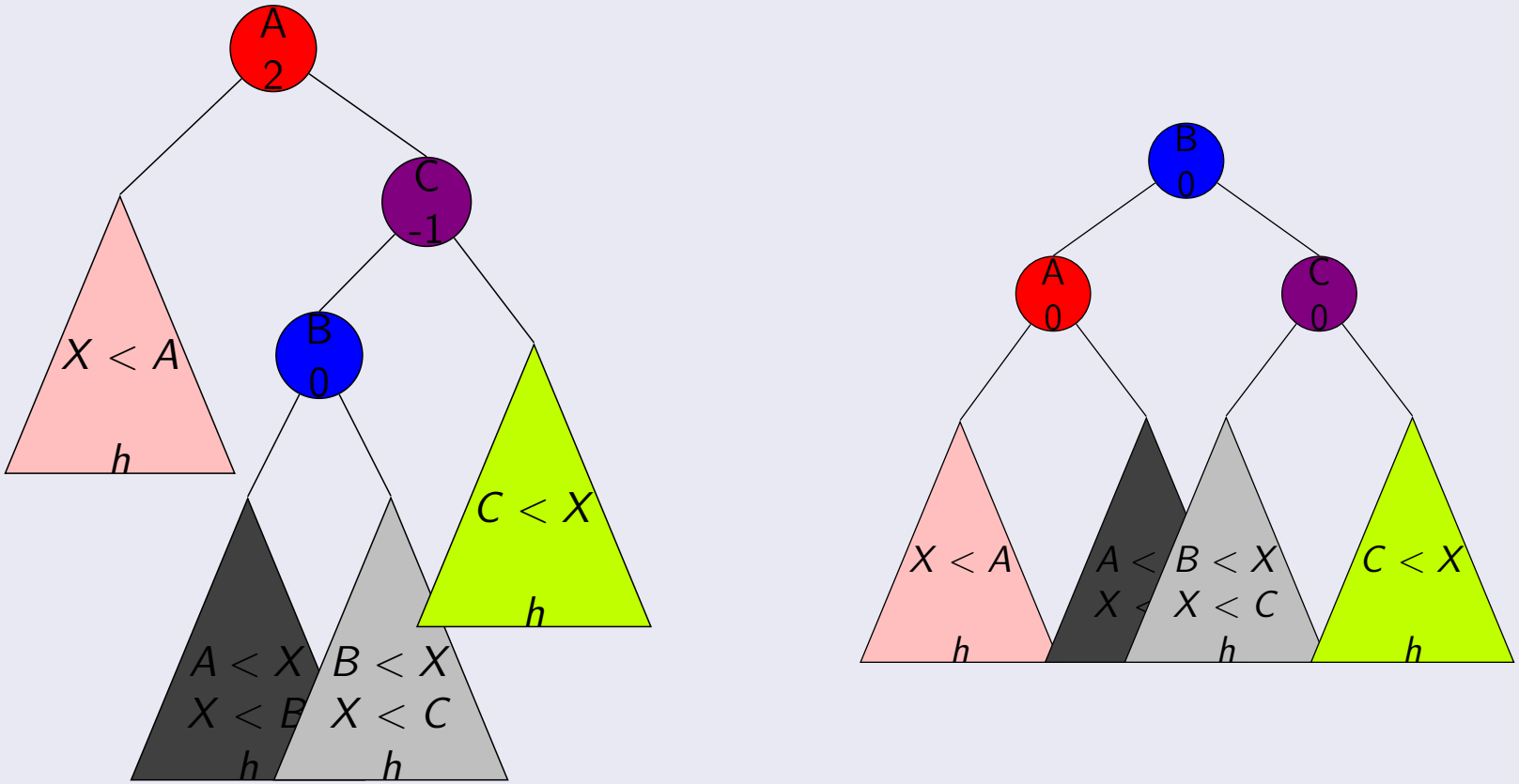
rotation 2/4: imbalance -2, imbalance left child -1



- simple right rotation

AVL Trees

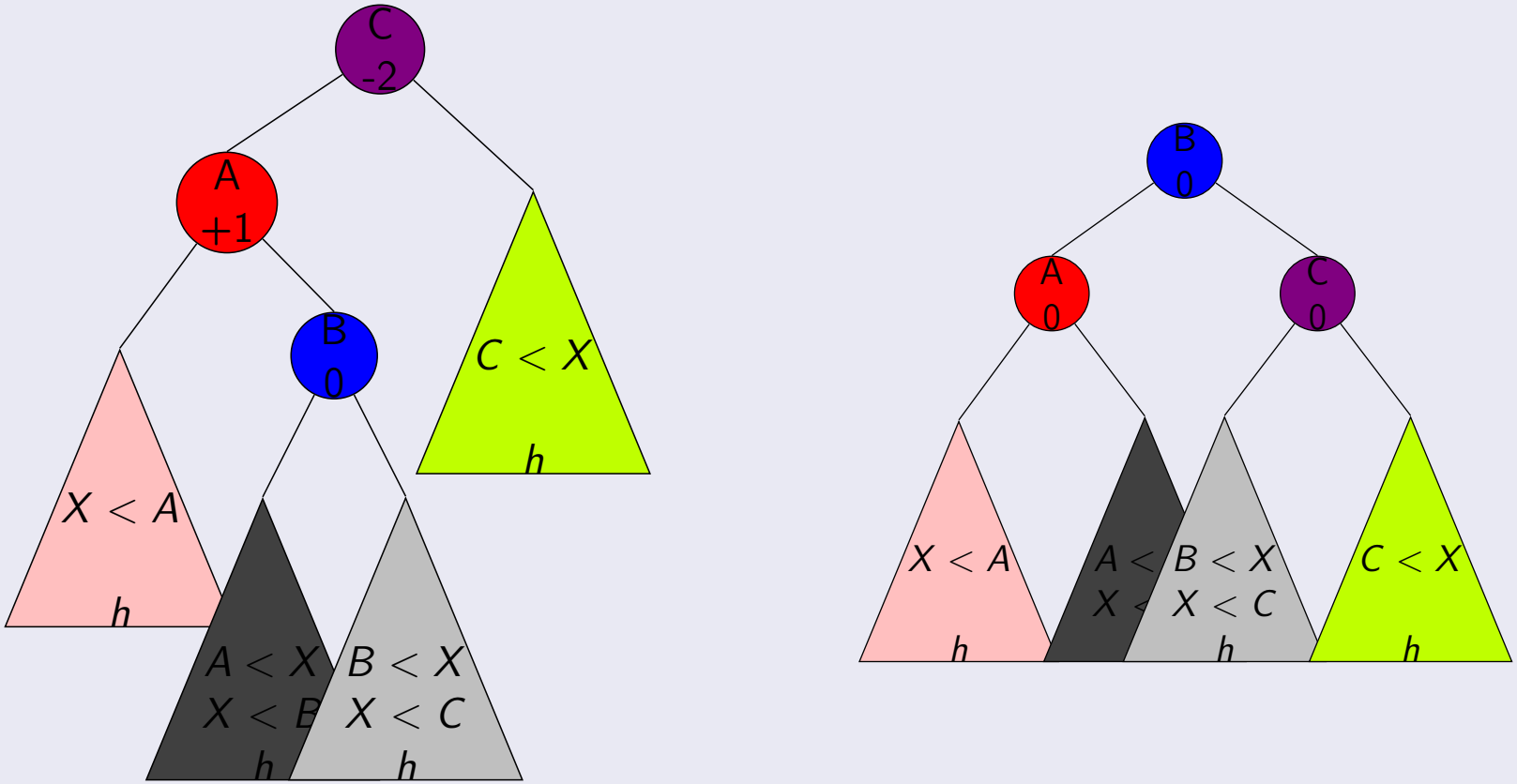
rotation 3/4: imbalance +2, imbalance right child -1



- double rotation left-right

AVL Trees

rotation 4/4: imbalance -2, imbalance right child +1



- double rotation right-left

AVL Trees

key observations

- a rotation reduces the height by 1
- insertion needs at most one rotation to rebalance
- removal may need $O(h)$ rotations to rebalance

AVL Trees

internal primitives

```
anode* anode_new_leaf(key k, value v);
anode* anode_delete_leaf(anode* n);
anode* anode_find(anode* n, anode_key k);
anode* anode_min(anode* n);
anode* anode_max(anode* n);
void anode_rot1(anode* lA, anode* A,
                anode* AB, anode* B,
                anode* gB);
void anode_rot2(anode* lA, anode* A,
                anode* AB, anode* B,
                anode* BC, anode* C,
                anode* gC);
```

AVL Trees

insertion

- Each insertion in the tree may imbalance it
- root r and new leaf n with path

$$r = p_0, p_1, \ldots, p_h = n$$

- update heights and balance along the path
- find largest $i \in \{0, \ldots, h\}$ such that

$$|height(left(p_i)) - height(right(p_i))| = 2$$

- perform the necessary rotation at p_i
- update heights and balance along the path
- insertion $\Longrightarrow O(h) = O(\log n)$

AVL Trees

deletion of a leaf

- root r and n the parent of the deleted leaf

$$r = p_0, p_1, \ldots, p_h = n$$

- for all $i \in \{h, \ldots, 0\}$
  - update height and balance p_i
  - if

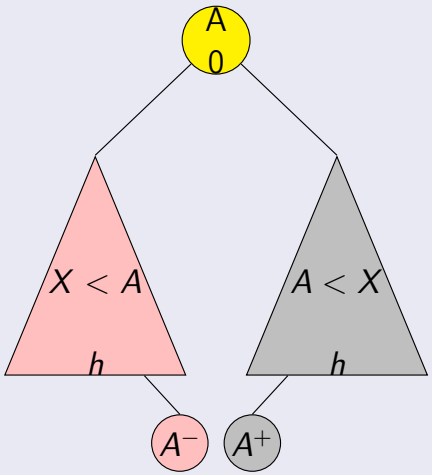
$$|height(left(p_i)) - height(right(p_i))| = 2$$

  - perform the necessary rotation at p_i

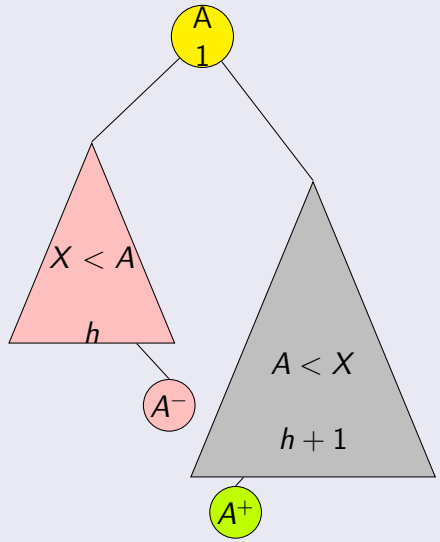
AVL Trees

deletion of an inner node n

- exchange n with "near" node
  - $A^- = \max(left(n))$
  - $A^+ = \min(right(n))$



balanced: any

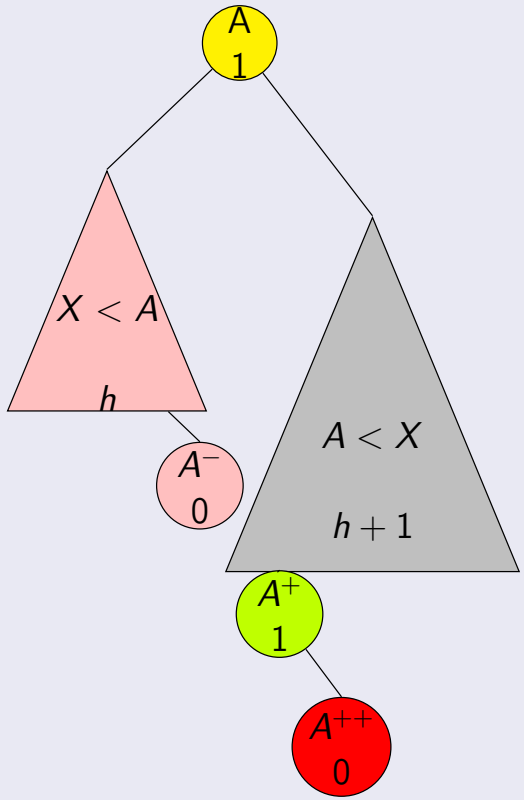


imbalanced: A^+

AVL Trees

deletion of an inner node n

- Difficulty: A^+ may not be a leaf

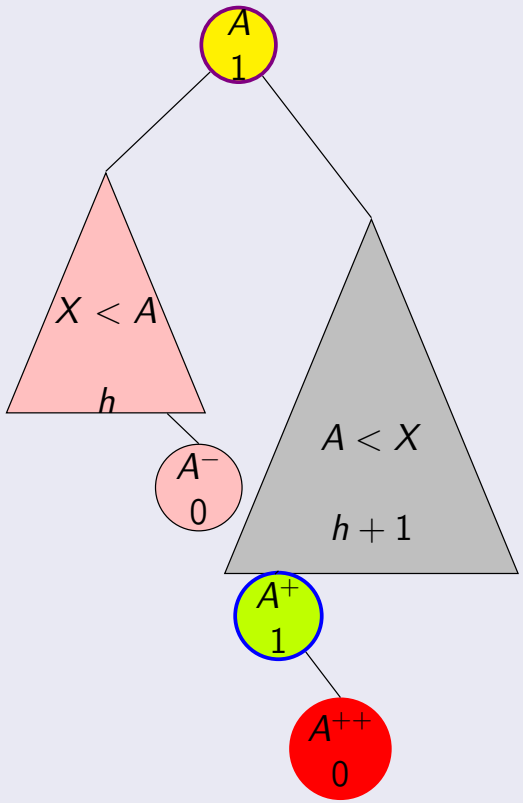


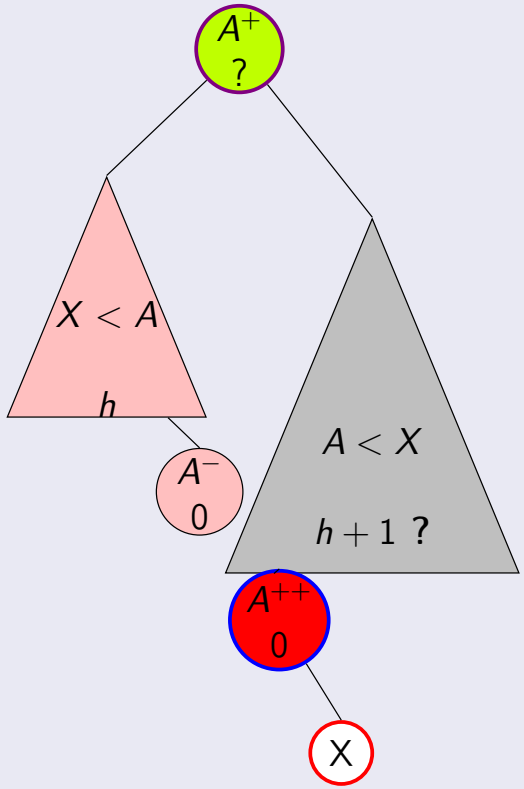
- AVL: A^+ has at most one child A^{++}
- AVL: A^{++} is a leaf

AVL Trees

AVL: deletion of an inner node n

- Copy contents of
  - $A^{++} \longrightarrow A^{+} \longrightarrow A$
- delete former node of A^{++}





Reminder: dictionaries

different levels of conception and development

1. algorithms and data structure design
2. encapsulation into abstract interface
3. definition of an Aplication Programmable Interface (API)
4. implementation (C programming)
5. management of the different components (make)
6. tests (validity)
7. benchmarks (efficiency)

Reminder: dictionaries

abstract interface dictionary type, `dict.h`:

```
dict* dict_create(void);
dict* dict_delete(dict* d);
size_t dict_size(dict const* d);
dict* dict_insert(dict* d, dict_key k, dict_value v);
bool dict_search(dict* d, dict_key k, dict_value* v);
dict* dict_remove(dict* d, dict_key k);
```

Reminder: dictionaries

implementing the dictionary with an AVL tree

`dict_create` $\leftrightarrow$ create a tree

`dict_delete` $\leftrightarrow$ delete a tree

`dict_size` $\leftrightarrow$ compute the size of the tree

`dict_insert` $\leftrightarrow$ search, insert, trace back and rotate

`dict_search` $\leftrightarrow$ search in the tree

`dict_remove` $\leftrightarrow$ search, search next or previous, exchange elements, trace back, rotate

Reminder: dictionaries

Steps:

- implement a search tree e.g `avl_tree.h` `avl_tree.c`
- don't modify `dict.h`
- encapsulate the tree functions through the `dict_` functions, `dict_avl.c`
- write a `main_dict.c` that only uses `dict.h`, not `avl_tree.h`
- link everything together `main_dict`: uses `main_dict.o`, `dict_avl.o`, `avl_tree.o`

Reminder: dictionaries

Pros and cons:

- \+ good deterministic complexity $O(\log n)$ for all operations
- \+ dynamic size
- – many pointers, many indirections
  - hard to program
  - hard to maintain
  - run time overhead
- – special (hidden) node data structure that depends on (key, value) type
  - memory overhead
  - memory leaks?

Table of Contents

Hash Tables

A completely different strategy

. . . to implement the same API

Dictionary search in constant time

- For very large n, $O(\log n)$ can be too large.
- Idea: use keys indices in a table, access in $O(1)$
- Problem: too many keys for a simple table
- Example: Dictionary of French
  $26^{25} > 10^{35}$ possible entries, when supposing that the longest word has 25 letters

Hash tables

Hashing

- Idea: regroup a set of keys to the same table index
- A hash function maps keys to integers

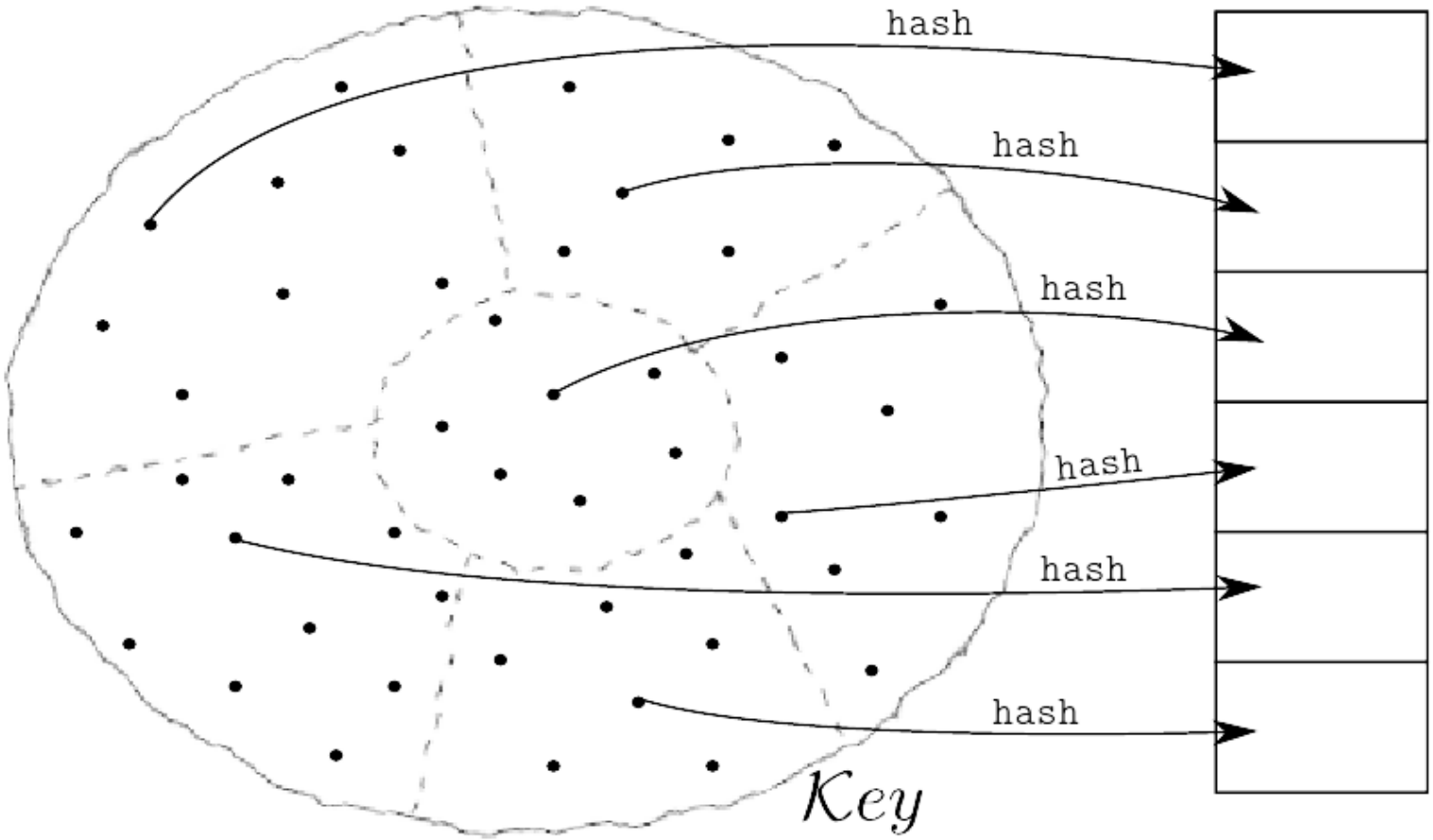
$$hash : keys \longrightarrow [0, \ldots, m-1]$$

where m is about the size of words to store in the dictionary

- The integers are typically the indices of a table T of m elements

$\rightarrow$ store a pair $(key, value)$ at the index $hash(key)$ in T.

Hash Tables



Hash Tables: Phone Diary Example

- Assume a inversed phone book: phone number $\mapsto$ name
- A hash function must reduce the key space to a smaller space: we might choose the hash function : $h(x) = x \ mod \ 4$
- Consider 03 88 27 10 13 $\rightarrow$ Jo,
- Consider 03 88 41 67 26 $\rightarrow$ Paul.

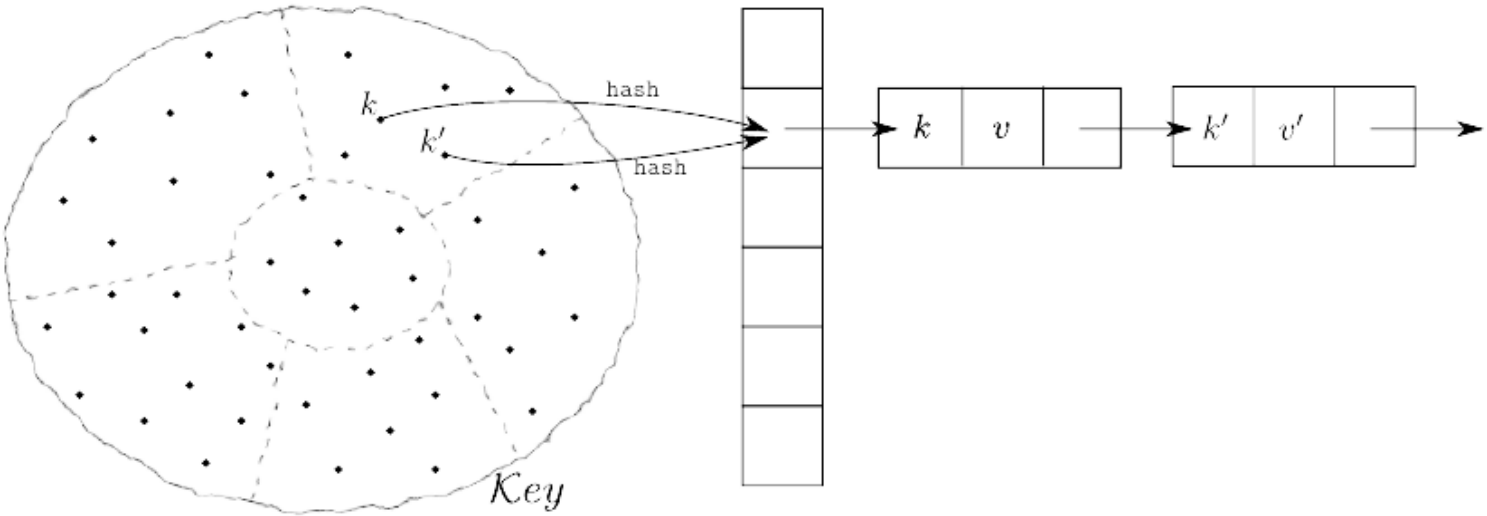
0	1	2	3
	Jo	Paul	

- Now, where would be stored Bob, with number 03 88 02 12 05 ? Answer: At $T[1]$, and this is a collision with Jo. Means we need a list in $T[1]$.

Hash Tables

Collision

- As there many more keys than table entries, *hash* can't be injective: eventually several keys for one table entry
- *Solution*: store an association list of `(key,value)` pairs



Hash Tables

Exemple

- 0388271013 $\rightarrow$ Jo
- 0388416726 $\rightarrow$ Paul
- 0388021205 $\rightarrow$ Bob
- 0070000001 $\rightarrow$ James

0	1	2	3
	Jo	Paul	
	Bob		
	James		

Hash Tables: Hash Function

What hash functions?

- to be better than lists, we have to *avoid collisions*.
- the choice of a good hash function is essential
- *French dictionary*
  If we just use the first two characters: ≈ 216 table entries
  - many duplicates, e.g starting with "ch"
  - few for other combinations, e.g "cj"

...	cf	cg	ch	ci	cj	ck	cl	cm	...
	1	1	616	226	0	0	184	1	

Hash Tables: Hash Function

choice of the hash function

- The quality of a hash function for a given *context* depends:
  - set of keys
  - the distribution of the keys
- *Goal:* we want uniform hashing
  - for every key k and all $i \in [0, \ldots, m-1]$
  - the probability for $hash(k) = i$ should be $\frac{1}{m}$

Hash Tables: Hash Function

Examples for good hash functions

- For uniformly distributed integer keys
  - division method:
    - Use $hash(k) = k \mod m$
    - *Problem:* needs that m is a prime number, far from a power of 2
  - multiplication method:
    - Let $0 < A < 1$ some transcendent number
    - Use the fractional part $f(k) = kA - \lfloor kA \rfloor$
    - Return the integer part of $\lfloor m \cdot f \rfloor$
    - Usually $m = 2^\ell$ for efficiency

      The value of $A = \frac{\sqrt{5}-1}{2}$ gives good results

Hash Tables: Hash Function

Examples for good hash functions

- $A = \frac{\sqrt{5}-1}{2} = 0.61803398875$, \$ m $= 2^{10} = 1024$\$
- example: keys are phone numbers
  - $0388271013 \rightarrow 239964682.880 \rightarrow 239964682 \rightarrow 522$
  - $0388416726 \rightarrow 240054738.467 \rightarrow 240054738 \rightarrow 466$
  - $0388021205 \rightarrow 239810293.046 \rightarrow 239810293 \rightarrow 757$
  - $0070000001 \rightarrow 43262379.831 \rightarrow 43262379 \rightarrow 427$

Hash Tables: Hash Function

Examples for good hash functions

- For strings
  - Problem: strings are not uniformly distributed
  - A commonly used hash function for strings looks as follows

```
uint64_t hash_string(size_t len, char const s[len]) {
  uint64_t ret = 0;
  for (size_t i = 0; i < len; ++i) {
      ret = ret*magic + s[i];
  }
  return ret;
}
```

where commonly used values for `magic` are 31, 33 and 37.

Hash Tables: Hash Function

Examples for good hash functions

- Example with `magic` $= 37$ and m $= 1024$

  $\texttt{zinc} = [122, 105, 110, 99] \rightarrow 122 \rightarrow 4514 + 105 \rightarrow 4619 \rightarrow$
  $170903 + 110 \rightarrow 171013 + 99 \rightarrow 171112$
  modulo $m \rightarrow 104$

  $\texttt{zone} = [122, 111, 110, 101] \rightarrow 6335796$
  modulo $m \rightarrow 308$

Hash Tables: Data Structure

list node

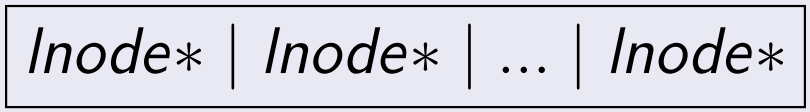
```
typedef struct lnode lnode;
struct lnode = {
  key key;
  value val;
  lnode* next;
};
```

- A list is given by an `lnode*`.
- We suppose that we have operations `lnode_push`, `lnode_pop`, `lnode_find`, ...

Hash Tables: Data Structure

dynamic table

```
typedef struct dict dict;
struct dict {
 size_t table_size;
 lnode** contents;
};
```

contents $\longrightarrow$ 

Hash Tables: Data Structure

table creation

We create a table of size `table_size`:

- each element is a list of `(key, value)` pairs.

```
dict* dict_create(size_t size) {
  dict* ret = malloc(sizeof *ret);
  ret->table_size = size;
  ret->contents = malloc(sizeof(lnode*[size]));
  for (size_t i=0; i<size; ++i) ret->contents[i] = 0;
  return ret;
}
```

Hash Tables: Insertion

element insertion

- compute $hash(k)$
- add the pair (k, v) top of the list at position $hash(k)$

```
dict* dict_insert(dict* d, key k, value v) {
 size_t h = hash(k) % (d->table_size);
 d->contents[h] = lnode_push(d->contents[h], k, v);
 return d;
}
```

Hash Tables: Search

element search

- compute $hash(k)$
- search a pair (k, v) in the list at position $hash(k)$

```
bool dict_search(dict* d, key k, value* v) {
 size_t h = hash(k) % (d->table_size);
 return lnode_find(d->contents[h], k, v);
}
```

Hash Tables: Removal

element removal

- compute $hash(k)$
- search a pair (k, v) in the list at position $hash(k)$ and remove it

```
dict* dict_remove(dict* d, key k) {
 size_t h = hash(k) % (d->table_size);
 lnode* start = d->contents[h];
 if (start) {
   lnode* prev = lnode_find_prev(start, k);
   if (prev) lnode_drop(prev); /* not first */
   else d->contents[h] = lnode_pop(start, 0, 0); /* first */
 }
 return d;
}
```

Hash Tables: Complexity

Complexity

complexity	insert	search	remove
average	O(1)	O(1 + α)	O(1 + α)
worst	O(1)	O(n)	O(n)

with $\alpha = \frac{n}{m}$

- worst case corresponds to the case of collisions
- for the average case, we suppose uniformity

Hash Tables: Resizing

dynamic resizing, description

- Consider n the number of elements to store, and m the table size:

To obtain a good average complexity, we may enlarge the table, typically if $n \gg m$.

- Create a new table T_2 of size $2m$
- Obtain a new hash function $hash_2$ on $[0, \ldots, 2m-1]$
- Insert the previous `(key, value)` pairs in the new table
  - for all pairs `(k, v)` in the actual table
    - compute $hash_2(k)$
    - insert `(k, v)` at position $hash_2(k)$ in T_2
  - delete the old table

Hash Tables

dynamic resizing, complexity

- copy cost for the whole $O(n + m)$
- copy cost amortized per entry O(1)
- by resizing $\alpha = \frac{n}{m}$ becomes a constant

complexity	insert	search	remove
average	O(1)	O(1)	O(1)
worst	O(1)	O(n)	O(n)

Dictionaries

Summary average complexity

	search	insert	remove
lists	$O(n)$	$O(1)$	$O(n)$
binary search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
AVL trees	$O(\log n)$	$O(\log n)$	$O(\log n)$
hash tables	$O(1)$	$O(1)$	$O(1)$

Summary worst case complexity

	search	insert	remove
lists	$O(n)$	$O(1)$	$O(n)$
binary search tree	$O(n)$	$O(n)$	$O(n)$
AVL trees	$O(\log n)$	$O(\log n)$	$O(\log n)$
hash tables	$O(n)$	$O(n)$	$O(n)$